Adyna: Accelerating Dynamic Neural Networks with Adaptive Scheduling

Zhiyao Li[†], Bohan Yang[‡], Jiaxiang Li[§], Taijie Chen[†], Xintong Li[†], Mingyu Gao[†¶$]
Tsinghua University[†], University of Science and Technology of China[‡], University of Toronto[§]
Shanghai AI Laboratory[¶], Shanghai Qi Zhi Institute[$]
{lizhiyao19, ctj19, lixt21}@mails.tsinghua.edu.cn, yangbh@mail.ustc.edu.cn
jxiang.li@mail.utoronto.ca, gaomy@tsinghua.edu.cn

***Abstract*—Dynamic architecture neural networks (DynNNs) are an emerging type of deep learning models that can leverage different processing difficulties of data samples to dynamically reduce computation demands at runtime. However, current GPUs and specialized accelerators lack the necessary architecture and dataflow support to achieve the promised theoretical efficiency improvements due to the high dynamism in DynNN execution. We propose Adyna, a novel hardware-software co-design solution to efficiently support DynNN inference. Adyna uses a unified representation to capture most existing DynNNs to enable a general design. It features a dynamism-aware, multi-kernel selection paradigm, in which the dataflow scheduler makes resource allocation decisions according to the distribution of dynamic size values, and the hardware architecture keeps multiple precompiled kernels and selects the best matching one to process each specific data sample according to its dynamic size. Adyna further uses an effective kernel sampling algorithm to carefully choose the set of kernels to load onto the hardware. Evaluated on various DynNN models, Adyna can outperform state-of-the-art multi-tile and multi-tenant accelerators by** $1.70\times$ **and** $1.57\times$ **on average, and up to** $2.32\times$ **and** $2.01\times$**.**

***Index Terms*—dynamic architecture neural network, domain-specific accelerator, scheduling**

I. Introduction

Deep neural networks (DNNs) are arguably the most important applications today [35]. Recently, a new type of DNNs named *dynamic architecture neural networks* (DynNNs) have emerged in computer vision and natural language processing [12], [14], [19], [42], [59], [70]. Instead of keeping a static specification of the model topology and the involved operators, DynNNs allow for dynamic decisions at runtime about what computations to execute for each input data sample, in order to best leverage the different difficulties of processing different data. Various DynNN proposals incorporate different types of dynamism in the model, including dynamic numbers of operators, dynamic operator sizes, and dynamic data processing paths in the model. By dynamically adapting the amount of computations to each data sample, DynNNs are able to save unnecessary computations and improve efficiency without sacrificing model accuracy [24]. Specifically, HydraNets [42] could reduce inference cost by $4\times$ with negligible accuracy loss; PABEE [70] is $1.62\times$ faster and has even better accuracy.

However, from a system perspective, the theoretical savings in computation demands of DynNNs may not realize practical performance improvements. DynNNs introduce significantly more dynamic runtime behaviors compared to conventional static DNNs, making it challenging to execute them efficiently in today's hardware. For example, when executed on GPUs, batching becomes inefficient for most DynNNs; hence most existing implementations are limited to batch-1 execution and underutilize the available parallel computing resources [59], [63], [70]. On one hand, the diversification of different data samples in the same batch leads to severe branch diversification [66], which is a well-known inefficiency. On the other hand, GPUs lack flexible control and usually need to ask the host CPU to make dynamic decisions, causing frequent CPU-GPU synchronization overheads.

Since DynNNs typically split data samples into smaller subsets that execute along different branches in the model, the computation load of each operator would decrease and cannot fully utilize the hardware resources. It is thus preferred to enable spatially co-located execution of multiple operators to share the chip. Three types of existing architectures have this potential, including multiplexable GPUs [43], [45], [60], multi-tenant DNN accelerators [22], [33], and multi-tile DNN accelerators [6], [18]. However, they all lack certain critical features or cannot support them efficiently for DynNN execution. Only multi-tenant designs can adjust resource allocation at runtime quickly enough to efficiently balance the highly varying dynamic workloads. Only multi-tile accelerators can directly forward data among multiple on-chip operators to reduce off-chip access overheads. None of them can efficiently select among multiple kernels for each operator that are optimized for different dynamic data sizes, or support flexible and dynamic on-chip communication that matches the input-dependent data splitting and routing in DynNNs.

In this work, we propose Adyna, a novel hardware-software co-design to efficiently support DynNN *inference*. We make the following technical contributions at multiple levels, *from algorithm representation, dataflow scheduling, to hardware architecture*, in order to address the aforementioned challenges and avoid the inefficiencies in existing GPUs and accelerators.

First, to support diverse DynNN types, we propose a novel *unified representation* that captures almost all known DynNN models, including dynamic model depths, dynamic operator sizes, and dynamic execution paths (Section IV). Our representation transforms all types of dynamism in the original computation graph onto the batch dimension, and introduces

2378-203X/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00049

a new *switch* operator to dynamically split the data samples in a batch. The unified representation enables a general and simplified hardware architecture design.

Second, we enhance the offline dataflow scheduling tools to be *dynamism-aware* (Section V). Instead of allocating resources and compiling dataflow schemes based on the worst-case maximum size, our scheduler uses a frequency-weighted approach based on the expectation of the computation load of each dynamic operator, which is calculated as the average size weighted by the occurrence frequency of each possible value. The occurrence frequencies are collected by a hardware profiler on our accelerator, and sent back to the software scheduler on the host CPU. We further propose optimizations to reduce the runtime instantaneous load variation, as well as the resource idleness for extremely rarely used operators.

Third, we add new hardware mechanisms to the state-of-the-art multi-tile DNN accelerators [17], [18], [50], [55] to better support DynNN processing (Section VI). The key idea is to keep *multiple kernels* that are optimized for different dynamic sizes in each compute tile, and dynamically select the best matching kernel for processing based on the actual size. To limit the on-chip storage of these many kernels, we implement a template kernel in the hardware control logic, and only store the metadata, such as loop dimensions and orders, to reduce the kernel size to only 128 bytes. We further modify the on-chip interconnect to support dynamic data routing and proper synchronization across the tiles.

Fourth, Adyna also features a novel multi-kernel sampling algorithm in its scheduler (Section VII), which effectively selects only a subset of kernels that are most likely to match the real execution distribution. This further restricts the number of kernels on the hardware and reduces the on-chip buffer size. Such kernel sampling also uses the runtime kernel occurrence frequency information collected by the hardware profiler.

We compare Adyna against the state-of-the-art multi-tile and multi-tenant DNN accelerators on various types of DynNNs that exhibit four types of dynamic behaviors. Adyna achieves 1.70× and 1.57× speedups on average, and up to 2.32× and 2.01×, over multi-tile and multi-tenant accelerators. A static version of Adyna with multi-kernel hardware selection and resource allocation contributes to an average 1.41× performance improvement. Further enabling dynamic scheduling adjustment at runtime brings another 1.21× speedup, and can approach the ideal case within only a 13% performance gap on average. We also demonstrate that the offline approach in Adyna is a better and more practical choice to support DynNNs. An alternative online approach that produces dataflow kernels in real time after knowing the actual dynamic sizes would need to finish kernel compilation within 0.39 ms to be competitive, which is an unrealistically high bar today.

II. Background and Motivations

A. Dynamic Architecture Neural Networks

Traditional deep neural network (DNN) models are mostly static, i.e., the computation graphs and the involved operators are fixed and must all be executed. *Dynamic architecture neural networks* (DynNNs) [12], [14], [19], [24], [28], [42], [59], [61], [70], in contrast, leverage the fact that input samples are not equal and require different processing efforts. They thus adapt the computation graph topologies and/or the operator shapes to each specific input sample, and dynamically save a significant portion of computations. For example in Figure 1, the bottom dog image is much easier to classify than the top one, allowing us to use fewer feature extraction layers to classify the simpler images.

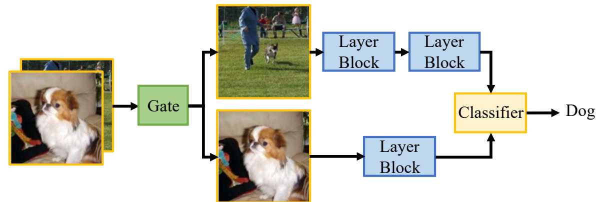

Fig. 1. An example DynNN model.

As shown in Figure 2, based on the taxonomy in [24], we categorize DynNNs into four typical types: *dynamic depth*, *dynamic width*, *dynamic routing*, and *dynamic region*. Dynamic depth networks (Figure 2(a)) vary on the number of layers each sample goes through [14], [32], [59]. When a sample reaches a special gate layer, the gate will make a prediction based on the current intermediate activation tensor and decide whether to continue extracting features or to directly output a result. Dynamic width networks selectively compute the channels of a layer [9], [19]. As in Figure 2(b), before feeding an activation tensor to a convolutional layer, the unimportant channels will be pruned. Such pruning is done dynamically on a per-sample basis, in contrast to static pruning. Dynamic routing networks (Figure 2(c)) route each sample to one or multiple branches. It is based on the assumption that different branches are trained as experts for different tasks [28], [29], [42]. Dynamic region networks restrict computations only to certain carefully selected informative regions. In Figure 2(d), only two out of the four regions (called patches) are actually used by the succeeding layer blocks. Some other DynNNs mix these four types for better performance. For example, AdaViT [40] combines layer skipping (dynamic depth) and patch selection (dynamic region). These four types cover most DynNNs in the survey [24] except element/neuron skipping networks. Element-wise dynamism/sparsity usually uses specialized data encoding (e.g., CSR and bitmap), and they are better supported by existing sparse DNN accelerators (Section X).

In this paper, we select five representative DynNNs listed in Table I as our workloads. For dynamic depth, SkipNet [59] and PABEE [70] use layer skipping and early exiting, respectively. For dynamic width, we choose FBSNet [19] that uses dynamic channel pruning. Tutel-MoE [28] is selected as a representative dynamic routing model, which is an implementation of Mixture of Experts (MoE). Lastly, for dynamic region, we use the differentiable patch selection (DPS) network [12]. These DynNNs cover the fields of computer vision (CV) and natural

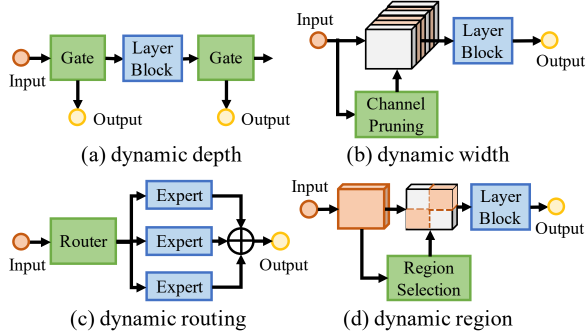

Fig. 2. Four typical categories of DynNNs.

TABLE I
EVALUATED DYNNNS IN THIS WORK.

Category	Network	Mechanism	Fields
Dynamic depth	SkipNet [59] PABEE [70]	Layer skipping Early exiting	CV NLP
Dynamic width	FBSNet [19]	Channel pruning	CV
Dynamic routing	Tutel-MoE [28]	Mixture-of-Experts	CV
Dynamic region	DPSNet [12]	Patch selection	CV, NLP

language processing (NLP) and are based on various backbone models including ResNet, BERT, and ViT [16], [26], [32].

B. Multi-Tile Accelerators and Dataflow

The importance of DNNs has made them an emerging domain for specialized hardware acceleration. Many existing DNN accelerators are derived from a common spatial architecture consisting of an array of processing elements (PEs) and a hierarchy of on-chip buffers including scratchpads and register files [8], [31]. As more complex and deeper DNNs require continuously increasing amounts of computations, recent DNN accelerators have to *scale* their computing capabilities. A common way to do so is to adopt a parallel, *multi-tile architecture* on a single chip or across multiple chips/chiplets [6], [17], [18], [50], [55]. As shown in Figure 3, a multi-tile accelerator connects a group of tiles and multiple off-chip memories using a network-on-chip (NoC). Compared with GPUs and spatial accelerators like TPUs where cores/PEs synchronize through off-chip memories, the most intriguing feature of multi-tile accelerators is inter-operator pipelining. By mapping consecutive operators (layers) to different tiles and directly forwarding intermediate data between tiles, pipelining could save off-chip memory transfers and improve performance.

With the abundant computing and data buffering resources offered by modern DNN accelerators, finding the best *dataflow* schemes to schedule various DNN layers to achieve high utilization becomes a critical task. Following prior wisdom [18], [47], [64], we abstract the dataflow scheduling of multi-tile accelerators into three levels: *graph segmentation*, *operator pipelining*, and *kernel generation*. At the graph segmentation level, the computation graph of the DNN model is partitioned

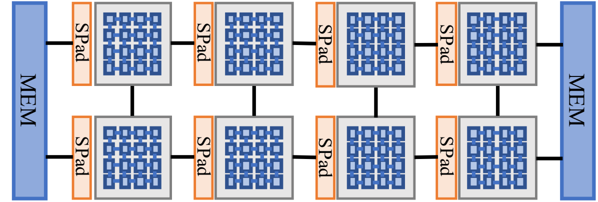

Fig. 3. A multi-tile DNN spatial accelerator. Each tile has an array of PEs, one register file per PE, and an SRAM scratchpad.

TABLE II
COMPARISON OF DYNNN SUPPORT ON CANDIDATE ARCHITECTURES.

Feature	GPU	M-tenant	M-tile	Adyna
F1. Spatial parallelism	✓	✓	✓	✓
F2. Fast runtime adjustment	✗	✓	✗	✓
F3. Operator pipelining	✗	✗	✓	✓
F4. Multi-kernel selection	✗	✗	✗	✓
F5. Dynamic data routing	✗	✗	✗	✓

into different groups (each called a *segment*) of consecutive operators. Segments are executed one after another on the accelerator. At the operator pipelining level, the operators in a segment are spatially scheduled to different tiles of the accelerator. The intermediate activation tensors are stored in the on-chip scratchpads and directly forwarded through the NoC between tiles in a pipelined manner, without swapping to memory. At the kernel generation level, we decide PE mapping, loop transformation, and parallelization across tiles for each operator. Such intra-operator schemes are commonly represented as nested-loop programs, which we call *kernels*.

C. Motivation

While DynNNs can reduce computations and data accesses, efficiently executing DynNNs is challenging with both GPUs and existing domain-specific accelerator designs. DynNNs introduce significant diversification among data samples in a batch, which effectively reduces the computation load of each operator and makes it more challenging for each individual operator to fully utilize the hardware resources in a large chip. In other words, instead of being processed by a single common operator, the samples in a batch are now diversified to multiple operators parallel to each other. Therefore, a highly desired architectural feature is to support spatial multi-operator parallelism, i.e., simultaneously running multiple operators on the hardware (**F1** in Table II).

There are mainly three candidate architectures with such spatial parallelism support, namely multiplexable GPUs [43], [45], [60], multi-tenant DNN accelerators (M-tenant) [22], [33], and multi-tile DNN accelerators (M-tile) [2], [6], [18], [69], as listed in Table II. Multiplexability allows a single GPU to be used simultaneously by multiple operators. This can be achieved through several ways, such as Multi-Instance GPUs (MIG) [10], [11], multiple CUDA streams [60], or CUDA dynamic parallelism (CDP) [45]. M-tenant can flexibly partition its computing and memory resources with various pre-

scheduled schemes to support multiple DNN models running simultaneously. M-tile can parallelize operators in one DNN in a pipelined manner as in Section II-B.

However, for efficient DynNN execution, these architectures all lack some necessary features or cannot support them efficiently (Table II **F2** to **F5**). First, DynNNs exhibit input-dependent computation load distribution among different operators, and thus the hardware resources that are allocated to these operators need to be fast and flexibly adjusted at runtime to adapt to the dynamic loads (**F2**). M-tenant could support fast runtime resource re-partitioning, after identifying which DNN models are currently running together [22], [33]. In contrast, multiplexible GPUs and M-tile cannot support sufficiently fast runtime adjustment. MIG needs up to several seconds to re-partition resources [36], [43]. The control flow between multiple CUDA streams requires frequent cross-stream synchronization, which needs to involve the host CPU with severe CPU-GPU synchronization overheads, up to 75% of the end-to-end latency [66]. While CDP supports runtime adjustment through conditionally invoking child kernels and allows dynamically determining the grid and threadblock sizes of each operator [45], it cannot fully avoid CPU-GPU synchronization and still suffers from data copy and kernel launch costs [7]. Besides, CDP treats the local and shared memories private in the parent and child kernels [45]. Thus intermediate data can only communicate through the global memory. Futhermore, CDP has additional overheads like runtime linking and register spilling [44], which may cause 20% slowdown [7], [30], [54], [58]. M-tile assumes static DNNs with known amounts of computations and pre-compiled mapping schemes [6], [18], lacking on-chip control to support runtime dynamism. For example, AMD XDNA [3], [52] is essentially an M-tile architecture and relies on the compiler toolchain for reconfiguration [3], which takes up to minutes and cannot serve real-time adjustment [4].

Furthermore, as DynNNs execute multiple operators together on-chip, inter-operator pipelining (**F3**) that directly forwards intermediate data among them is a natural choice to reduce off-chip accesses. Only M-tile, e.g., XDNA [52], supports it by design. Multiplexable GPUs and M-tenant both assume the co-executing kernels are loosely related, e.g., from different CUDA streams or different DNN models, and can only communicate through the global memory.

Finally, there are DynNN-specific requirements that none of the three architectures supports. When we look at a single DynNN operator, usually one of its dimensions is dynamically determined at runtime (e.g., batch for dynamic depth, channel for dynamic width), which leads to varying shapes and favors different optimized dataflow schemes, i.e., kernels. Using a large one-size-fit-all kernel will lead to suboptimal and sometimes redundant computations and memory accesses. Consequently, each operator should be associated with multiple kernels that the hardware needs to store on-chip and select dynamically at runtime (**F4**). Moreover, DynNNs usually fork into and merge from multiple branches with irregular and input-dependent data division and reduction [12], [28], [59].

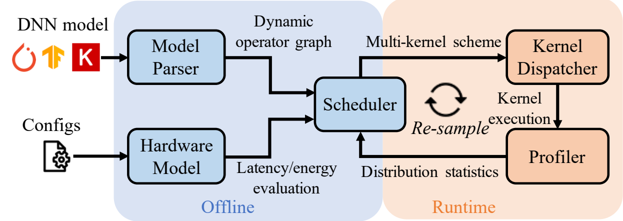

Fig. 4. Adyna overview and its overall workflow.

This requires flexible and efficient on-chip data routing support in the hardware to properly forward data between operators and eliminate redundant data transfers (**F5**).

III. Adyna Overview

We propose Adyna, a hardware-software co-design solution for DynNN model *inference*. Adyna is optimized for DynNNs and supports all missing features of previous architectures in Table II. The overall workflow of Adyna is shown in Figure 4.

First, Adyna uses a novel representation called *dynamic operator graphs* to handle various DynNN types in Table I in a unified manner (Section IV). The DynNN model is first processed by the model parser to analyze its operators, dependencies, and routing policies to construct the dynamic operator graph. Instead of having all diverse dynamic behaviors in Figure 2, the dynamic operator graph unifies them into a single type with only dynamic sizes on the batch dimension, serving as a unified foundation for our hardware and software designs.

The dynamic operator graph is then fed into the scheduler. The scheduler searches for optimized dataflow schemes with a hardware cost model [8], [34], [64]. The scheduler leverages a new *dynamism-aware multi-kernel execution* paradigm to enable fast runtime adjustment (**F2**). It combines statistic-based offline scheduling and real-time online adjustment, to deal with long-term and short-term dynamic variations, respectively (Section V). This achieves a good tradeoff between scheduling cost and load imbalance. Specifically, Adyna collects the distributions of runtime dynamic values using a hardware-based profiler. It periodically calculates the expectation of each operator's workload and accordingly allocates resources. To further alleviate real-time load imbalance at runtime, Adyna adopts two additional techniques: *tile sharing* to adjust resource partitioning between branches, and *branch grouping* to improve the scheduling of related and sparse branches.

The scheduled dataflow schemes are loaded to the hardware to execute. The Adyna hardware (Section VI) is based on multi-tile accelerators with inter-operator pipelining (**F3**), and extended to further support *multi-kernel selection* of each operator (**F4**) and *dynamic data routing* between operators (**F5**). For each operator, Adyna uses a space-efficient format to maintain multiple kernels with different allocated resources, optimized for different dynamic dimension sizes. When encountering a concrete size at runtime, the best-matching kernel is dynamically selected by a kernel dispatcher and interpreted as instructions to orchestrate computations. Between operators,

the hardware features an enhanced NoC interface for flexible data routing across tiles.

Finally, we notice that the possible sizes of the dynamic dimension in an operator, and correspondingly the number of kernels, could be quite large (hundreds to thousands). This makes it impossible to store all of them on-chip. Thus Adyna proposes a *multi-kernel sampling* technique (**F4**). Based on the collected value distribution of the dynamic size, the scheduler decides which kernels among all the possible ones are most likely used at runtime and should be compiled and stored in the accelerator (Section VII). As observed in previous works [13], [25], the value distribution can change over time. To be adaptive to dynamic shifts in the value distribution, such kernel selection is also periodically re-sampled by the scheduler after receiving the report from the hardware profiler.

IV. Unified Representation

Different types of DynNNs cause different dynamic behaviors in graph topologies and operator shapes, which makes it difficult to perform static scheduling during compile time. In Adyna, we propose a novel approach to transform these diverse types of dynamism into a unified representation called *dynamic operator graphs*, enabling us to handle all DynNNs in the same way, simplifying the software and hardware designs.

To capture the fact that different data samples may take different execution paths, we first introduce a new *switch* operator. A switch operator is configured to switch the input tensor along a given dimension *dyn_dim* into multiple output tensors for the succeeding selectively executed operators along different branches. As shown in the center of Figure 5, the switch operator receives an input tensor and a routing mask. The mask is generated using other operators in the DynNN model, e.g., fully connected operators in the figure. It represents which branches are activated for each sample in the batch. For example, the routing mask in Figure 5 indicates that sample 0 activates branches 0 and 1, sample 1 only activates branch 1, and sample 2 activates branch 2. Based on the routing mask, the input tensor is split along *dyn_dim* and routed to different succeeding operators. Compared with conventional tensor split [46], the switch operator allows for non-sequential and non-uniform tensor splitting and broadcasting across multiple divisions, while conventional split only follows a sequential, predefined manner.

Now we could keep all selectively executed operators in the computation graph of a DynNN as multiple branches after a switch operator, and maintain a static topology. We call such a transformed graph a *dynamic operator graph*. When defining a DynNN model, the user only needs to use the customized switch and merge operators to specify the dynamic structures, while still reusing existing APIs from mainstream frameworks for the rest static operators. For example, to use a switch, the user should specify the dynamic dmension, as well as both the input data tensor and the routing mask generated from standard `conv2d` or `matmul` operators. The operator will generate a dynamic tensor object which should connect to all the succeeding operators. Under the hood, the model parser automatically tracks the propagation of dynamic dimensions, and analyzes the operators, dependencies, and routing policies to construct the dynamic operator graph. Existing DNN frameworks lack a standard way to implement various DynNNs, and usually require users to write low-level `if-else`-like code structures. Thus we expect the new switch operator is easy to use and does not complicate programming.

However, the challenging part is that different DynNN types have different *dyn_dim*s, complicating hardware and software support. Our key insight is that we are able to transform all types of dynamism in various DynNNs *only onto the batch dimension*, thus enabling unified designs. Figure 5 illustrates how we are able to represent *all* the DynNNs listed in Table I in such a way. (a) Early exiting can be represented using a switch operator to either a sink or the subsequent operators. (b) MoE is a switch operator followed by multiple branches. To represent channel pruning, we divide the convolutional operator into several sub-operators along the input channel dimension, and treat each as a separate branch selected by the switch operator. Each sub-operator is a dense operator but with a dynamic size. (c) Layer skipping can be represented by adding a shortcut beside the backbone operators after the switch operator. (d) For patch selection, we fold the iteration over patches into the batch dimension, and use a sink operator to discard those unimportant patches.

Note that the selectively executed branches after the switch operator may have drastically varying values in their *dyn_dim* (i.e., batch) sizes (some could even be empty), which leads to imbalanced computation and memory access demands. This dynamism will be passed to all subsequent operators until a merge point. All such operators with dynamic batch sizes are marked with shadows in Figure 5. To precisely characterize the dynamism, we further enhance each of these operators to *track the value frequencies* of their *dyn_dim* sizes when executing on hardware, as shown in Figure 5 left. This frequency track table will be filled in by the hardware profiler and used by the software scheduler.

To construct a dynamic operator graph, the model parser inserts necessary switch operators and makes certain transformations similar to Figure 5. It then traverses the graph to convert corresponding operators (e.g., those between switch operators and merge points) to dynamic, i.e., adding a *dyn_dim* and initializing a frequency track table in software. Currently, we only allow each operator to have at most one *dyn_dim*. We disallow one operator on multiple branches of a switch operator unless it is a merge with a matched *dyn_dim*. We also disallow one operator to be the common successor of two switch operators, unless one switch operator is the successor of the other. Our representation is expressive enough to cover both the four basic types of DynNNs introduced in Section II-A as well as their hybrid ones like AdaViT [40].

V. Dynamism-Aware Scheduling

The dynamic operator graph of a DynNN is processed by the Adyna scheduler to determine the optimized resource allocation and dataflow schemes. Similar to the static cases,

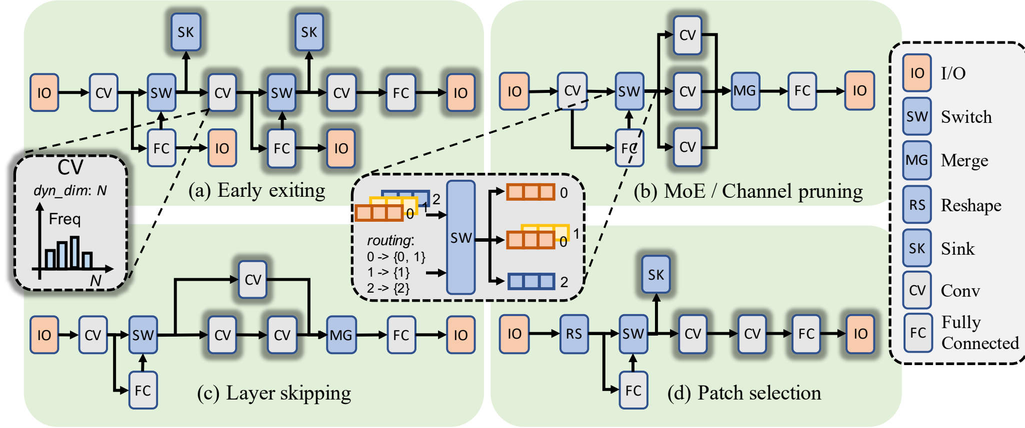

Fig. 5. Dynamic operator graph representation. Operators with dynamic dimensions are marked with shadows. Non-linear operators are omitted for simplicity.

our scheduler also works on the following three levels: graph segmentation, operator pipelining, and kernel generation.

A. *Frequency-Weighted Segmentation and Allocation*

The common practice of mapping a DNN model onto an accelerator is to first partition the model into multiple segments, each of which contains multiple operators to be executed on-chip simultaneously in a pipelined way. Each operator is allocated a certain number of accelerator tiles. Determining optimized segmentation and pipelining schemes requires information on the operator shapes and their dependencies [18], [55]. Such size information is trivial to obtain in static DNNs. But for DynNNs, these sizes are only known at runtime. A naive way is to assume the maximum (i.e., worst-case) value for each dynamic dimension (*dyn_dim*). Figure 6 top illustrates a layer skipping block with two branches, B1 with one convolutional operator and B2 with two. The total *dyn_dim* size is 8. To schedule on an 8-tile accelerator, the static approach assumes the same *dyn_dim* size for both branches to derive the computation demand ratio, which is $1:2$ for B1 vs. B2. So 3 tiles are allocated to B1 and 5 to B2.

However, if we analyze a real execution trace, e.g., doing inference of SkipNet [59] on ImageNet [15], we find that the actual *dyn_dim* values on the two branches are quite different. As in Figure 6 bottom, the horizontal axis represents a series of batches. Each bar represents the normalized per-tile workload of a batch, assigned to either branch. In the static allocation case, on average 5.03 samples out of 8 in a batch use B1, while the rest 2.97 follow B2. Consequently, in most batches, B1 suffers from more workloads per tile than B2 in Figure 6(a) if using the static allocation strategy. Such load imbalance causes resource underutilization and degrades performance.

To solve this, we propose *frequency-weighted segmentation and allocation*. Instead of using the maximum value for segmentation and allocation, we use the expectation of the *dyn_dim* value based on its collected distribution from the on-chip profiler. The demand resource is allocated proportional to this expectation. Back to the prior example, the *dyn_dim* expectations of B1 and B2 are 5.03 and 2.97, respectively,

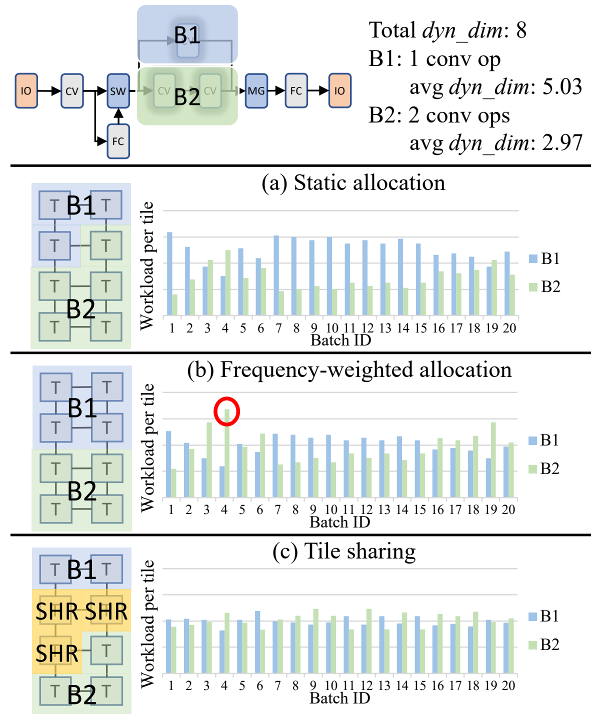

Fig. 6. Frequency-weighted allocation and tile sharing. The trace shows the workload per tile of each batch for two branches B1 and B2.

while B1 has 1 operator and B2 has 2. So their resource ratio should be $(1 \times 5.03) : (2 \times 2.97)$, approximately 4 tiles allocated to each. Figure 6(b) shows that the frequency-weighted strategy achieves a more balanced workload per tile.

B. *Optimizations for Runtime Adjustment*

Tile sharing. While frequency-weighted allocation improves load balance in the average case, the *dyn_dim* values can still exhibit significant variations for each specific batch and diverge from the calculated expectation. Figure 6(b) high-

lights one such case with a red circle, in which B2 temporarily has more workloads than expected. To deal with this runtime variation, we further propose the *tile sharing* optimization, which allows one tile to be allocated to multiple operators on different branches succeeding the same switch operator. At runtime, when the actual *dyn_dim* values are known, we decide which operator to execute on this shared tile to achieve the best balance. Essentially, tile sharing is one kind of work stealing. Our hardware design in Section VI supports such capability of dynamically selecting which kernel to execute.

A key point is that we must limit the degree of sharing to avoid excessive on-chip storage overheads. In the extreme case, all tiles can be shared by all branches, but this would result in too many kernels buffered in each tile. It is also unnecessary because of the diminishing improvements as sharing increases. We apply the following restrictions. First, we only share a tile among at most two branches, which are the two branches that are least likely to be activated at the same time. This helps complement their resource requirements and reduce contention. Second, we decide the number of shared tiles by only supporting three cases: the ratio of the two branches' average computation demands $a : b$, and two nearby ratios $2a : b$ and $a : 2b$. As in Figure 6(c), tile sharing leads to three ratios of $5.03 : 5.94$, $10.06 : 5.94$, and $5.03 : 11.88$, corresponding to tile allocations of $4 : 4$, $5 : 3$, and $2 : 6$, requiring 3 shared tiles. From the trace we see that such tile sharing can substantially reduce most workload variations, especially the temporary spikes.

Branch grouping. In some cases, e.g., FBSNet [19], the loads of the branches are highly skewed, i.e., some branches are not even executed once in a batch, or only at very low frequencies. In these cases, the tiles allocated to the unexecuted branches are underutilized. We thus use *branch grouping* to group these low-frequency branches together and execute them on the same tiles temporally. For example, if two branches are rarely executed, i.e., their activated frequencies are below a predefined threshold, we use the sum of their expected workloads for allocation and use the allocated tiles for both branches temporally. The tile hardware stores the kernels of both branches and executes them as needed (Section VI). This is different from the spatial allocated tile sharing described before, where different branches always use a disjoint set of tiles. Grouped branches temporally use the same tile.

C. Scheduling Overheads

Dynamism-aware scheduling introduces small overheads for runtime performance. The frequency-weighted segmentation and allocation are performed peridoically and would cause hardware reconfiguration including pipeline draining, but this happens in a coarse granularity (e.g., every 40 batches in Section VIII) and incurs less than 2.4% overheads in our evaluation. The tile sharing and branch grouping techniques are achieved through pre-compiled kernels as introduced in Section VI. During runtime, only the routing mask matching in switch operators and the multi-kernel selection are needed, which have negligible time cost.

VI. Hardware Architecture

The Adyna hardware is optimized on top of the baseline multi-tile architecture. We support multi-kernel selection inside each tile by adding a kernel dispatcher and carefully managing the space to buffer multiple kernels (Section VI-B). We also enhance the NoC to dynamically split and route data and synchronize execution progress across tiles (Section VI-C).

A. Overall Architecture and Execution Flow

As in Figure 7 left, the tiles are placed in a 2D layout, connected through a 2D-torus NoC (for simplicity the loop-back connections are not shown in the figure) with on-chip routers to transfer data across tiles. We adopt X-Y routing to avoid deadlocks. Each tile can also reach the closest memory interface for off-chip data accesses.

Figure 7 right shows the tile design. Each tile consists of a PE array and a scratchpad as in the baseline. It adds three components specially designed for DynNNs: a kernel dispatcher, a network interface, and an integrated controller and profiler. The *kernel dispatcher* interprets the runtime *dyn_dim* value and dynamically selects the most appropriate kernel to execute. The *network interface* supports dynamic data splitting and routing for the switch operators, and prepares the data packets sent to the router. The *profiler* tracks the *dyn_dim* value distributions and periodically reports to the scheduler for resource allocation and kernel re-sampling (Section VII).

Similar to the baseline multi-tile architecture, at runtime, each operator is mapped to multiple tiles, so that each tile processes a partition of an operator, by executing a specific kernel compiled by the scheduler and locally stored in the tile. The kernel execution is triggered by the arrival of input data from the NoC. Each tile is also made aware of the information of its succeeding operators, including which tiles are allocated to them, and how the current output data should be partitioned among these tiles. Hence the tile knows how to send its output data to the other tiles through the NoC.

B. Multi-Kernel Selection in a Tile

To support dynamic data sizes at runtime, each tile allocated to a dynamic operator would keep multiple kernels optimized for different *dyn_dim* values. These kernels are stored in the scratchpad, sharing the same buffer space with the data. At runtime, when the input data arrive, the metadata in the first packet contain the actual *dyn_dim* value, which is used by the kernel dispatcher to search among the locally stored kernels and select the best matching one, i.e., the kernel with the smallest *dyn_dim* value that is no less than the actual value. The profiler tracks the invocation frequencies of all the kernels, and periodically sends the distribution to the scheduler.

The key challenge to supporting multi-kernel execution is the limited on-chip space to store the many different kernels in a tile. In our current design, the scratchpad in each tile has 512 kB. To avoid interference with data buffering, we restrict the maximum capacity occupied by the kernels to be less than 5% of the scratchpad, i.e., 25.6 kB. This is a rather small

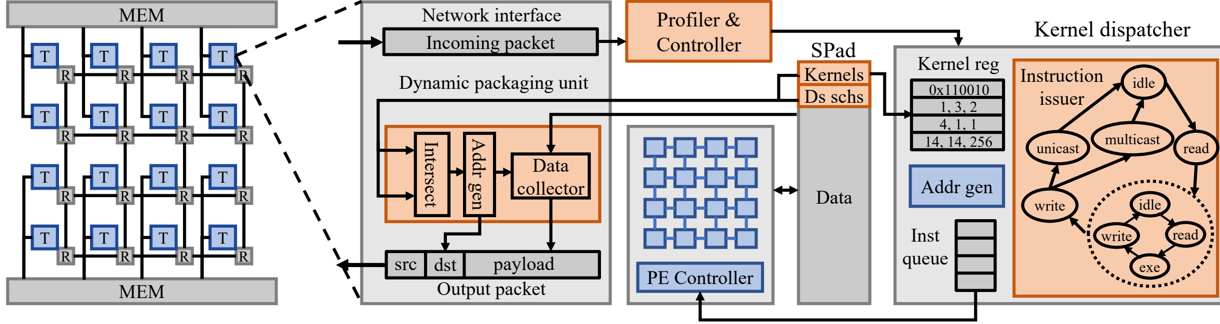

Fig. 7. Adyna hardware architecture. The left side shows the overall multi-tile accelerator. The right side shows a single tile. Each tile contains a PE array, a scratchpad, a network interface, a tile controller, and a kernel dispatcher. New components in Adyna are highlighted in orange.

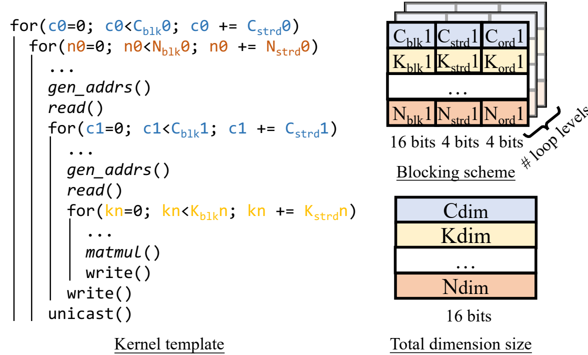

Fig. 8. The pseudocode of kernel template and the metadata format. $C_{blk}i$, $C_{strd}i$, $C_{ord}i$ are loop C's blocking factor, iteration stride, and order at level i.

instruction buffer, which limits the number of kernels each tile can support as well as the size of each kernel.

Adyna uses two techniques to address this issue. First, the scheduler conducts *kernel sampling* to carefully select which kernels to buffer in each tile to restrict the number of kernels, which will be discussed in Section VII. Second, we use *template kernels* to reduce the size of each kernel. Figure 8 shows the generic kernel template, which is essentially a classical nested loop structure that reads input data, does matrix/tensor multiplications, and writes output data back. The number of loop levels matches the number of memory hierarchy levels, following DNN loop blocking techniques [34], [47], [64]. Each blocked data dimension at a loop level contains the blocking factor (16 bits), the iteration stride (4 bits), and its order at this level (4 bits). We also specify the total size of each dimension, e.g., Cdim (16 bits). In this way, we only need to store these small metadata to represent a kernel, instead of a large, full program. For 7 dimensions and 5 loop levels, such kernel metadata only take about 128 bytes.

The kernel dispatcher implements this generic template as a finite state machine in hardware, i.e., the instruction issuer in Figure 7. After loading the metadata of a kernel, the instruction issuer iterates over the loops and generates corresponding instructions. We use an address generator to calculate the data access addresses from the blocking factors and strides. The instruction issuer is also responsible for transferring the output data to the next tiles, for which it issues unicast or multicast instructions to the network interface.

The instruction issuer is also able to do *runtime kernel-fitting* to further eliminate unnecessary computations and data movements caused by the gap between the kernel and the real *dyn_dim* value. It compares the current loop indices with the actual loop bounds determined by the real *dyn_dim* value. If exceeded, it will not generate the instructions to fetch the data and skip the computations.

Our kernel templates support common element-wise and in-place kernel fusion, e.g., pooling and layer normalization, by optionally issuing corresponding instructions in the predefined nested loops. Arbitary kernel fusion is not supported, as known to be a common limitation of domain-specific hardware that has fixed instruction sets [8], [18], [31].

C. Dynamic Data Routing across Tiles

The dynamic execution in DynNNs also results in varying communication patterns on the inter-tile NoC. We introduce the network interface unit in each tile to support *dynamic data routing* and *efficient synchronization* across tiles.

To support dynamic data routing, the hardware specially handles the switch operator. It is fused into the tiles of its preceding operator who generates the tensor to be dynamically split. These tiles will use the routing mask computed elsewhere to configure their network interface units. As shown in Figure 7 middle, the network interface is responsible for setting the routing destinations and packing the data. To accurately route the right range of data to the correct destination, the controller first infers the best-fit kernel of each succeeding operator, based on the actual *dyn_dim* value it has. From these concrete succeeding kernels we know the exact data layouts in the destination tiles. Then, we derive the range of the output tensor to be sent to each tile, by applying the succeeding operator's intra-operator parallelization scheme to the output tensor dimension range. Finally, we use the desired data layout of the destination tile to generate the sequence of data addresses, which are used by the data collector to fetch data from the local buffer and send out through the NoC.

Different from static on-chip pipelining execution with a fixed latency at each stage, the dynamic data sizes may cause the pipeline stages to exhibit varying execution times. Proper synchronization between adjacent stages is thus needed. When a tile finishes a kernel and is about to transfer the output data to another tile, it will first send a special probing packet to query the destination tile. The destination tile only acknowledges the probing packet with a response when it is ready to accept the data in its local scratchpad. The source tile does not start transferring data until it receives the acknowledgment.

VII. Multi-Kernel Sampling

As Section VI-B discussed, we need to restrict the number of kernels in each tile. Given the allowed kernel storage space 25.6 kB and the size of each kernel 128 bytes, we can at most store 200 kernels in each tile. Also considering the tile sharing in Section V-B amplifies the number of kernels by a factor of 6 (sharing 2 operators with 3 allocation ratios), the maximum kernel count is about 32. On the other hand, both the range and the variance of *dyn_dim* distributions can be large. For example, DPSNet [12] folds its dynamic dimensions into the batch dimension, further increasing the *dyn_dim* size up to 8192, making it impossible to keep all the kernels on-chip; meanwhile, the variation in its patch selection is also very large since the objects can be in arbitrary regions. The scheduler thus needs to determine which subset of kernels to buffer in each tile to maximize the overall efficiency. It makes these decisions based on the *dyn_dim* value frequency distribution periodically reported by the hardware profiler. Essentially, we prefer to keep the kernels optimized for the most frequently occurring *dyn_dim* values, while sacrificing the efficiency of the rare values. We call this process *multi-kernel sampling*.

Algorithm 1 Multi-kernel sampling

Require: Sampled values of *dyn_dim* `vals[]`, and their frequencies `freq[]`.
Ensure: Re-sampled values `new_vals[]`.
1: `new_vals` ← `vals`; `new_freq` ← `freq`
2: **for** N iterations **do**
3: ▷ Remove the value with the least punishment.
4: `punish` ← CalcPunish(`new_vals`, `new_freq`)
5: `rm_pos` ← ArgMin(`punish`)
6: `rm_val` ← `new_vals`.Remove(`rm_pos`)
7: ▷ Add the value with the most saving.
8: `saving` ← CalcSaving(`new_vals`, `new_freq`)
9: `in_pos` ← ArgMax(`saving`)
10: `in_val` ← (`new_vals`[`in_pos` − 1] + `new_vals`[`in_pos`])/2
11: **if** `in_val` = `rm_val` or `in_val` is invalid **then**
12: ▷ Recover the removed value.
13: `new_vals`.Insert(`rm_pos`, `rm_val`)
14: **break**
15: **else**
16: `new_vals`.Insert(`in_pos`, `in_val`)
17: ▷ Redistribute the frequencies (Algorithm 2).
18: `new_freq` ← RedistFreq(`vals`, `freq`, `new_vals`)
19: `vals` ← `new_vals`; `freq` ← `new_freq`

Initially, we start with a set of kernels uniformly spanned between 1 and the maximum *dyn_dim* value. During execution, after the profiler sends a copy of the tracked frequency distribution back to the scheduler on the host CPU, the scheduler re-samples a new set of kernels that better match with the

Algorithm 2 Redistribution of frequencies

Require: Sampled values of *dyn_dim* `vals[]` and their frequencies `freq[]`, re-sampled values `new_vals[]`.
Ensure: Redistributed frequencies `new_freq[]` for re-sampled values.
1: Initialize `new_freq` to all 0 with same length as `freq`
2: **for** `pos` in 0 to Len(`freq`) **do**
3: f ← `freq`[`pos`]
4: ub ← `vals`[`pos`]
5: **if** $ub <$ `new_vals`[0] **then**
6: Add f to `new_freq`[0]
7: **continue**
8: lb ← LowerBound(`vals`, `pos`)
9: `samples` ← `new_vals`.Filter(lambda $x : x > lb \wedge x \leq ub$)
10: $pv \leftarrow lb$
11: **for** p, v in Enumerate(`samples`) **do**
12: $f' \leftarrow f \times (v - pv)/(ub - lb)$
13: Add f' to `new_freq`[p]; $pv \leftarrow v$

runtime distribution. Algorithm 1 summarizes the sampling algorithm for a dynamic operator, which takes in the current set of sampled *dyn_dim* values (sorted) and their frequencies, and outputs a new set of re-sampled values. The algorithm executes iteratively. In each iteration, we try to remove a value and insert a new one, by evaluating the punishment of removing a sample value and the saving of inserting a new value (Lines 4 and 8). To define the punishment and the saving, we use the difference between the actual *dyn_dim* value $\hat{v}$ and the best matching sample value v_i (the closest one that is no less than $\hat{v}$), $v_i - \hat{v}$, which represents the loss of using this kernel for this *dyn_dim* value. Suppose the real frequency distribution of v is $\phi(v)$. In the range of v_{i-1} to v_i, the total loss is $\sum_{v=v_{i-1}}^{v_i} \phi(v)(v_i - v)$. If we remove v_i, the *dyn_dim* values in the above range need to instead use the kernel for v_{i+1}. So the loss increases by

$$\begin{aligned}\Delta(v_i) &= \sum_{v=v_{i-1}}^{v_i} \phi(v)(v_{i+1} - v) - \sum_{v=v_{i-1}}^{v_i} \phi(v)(v_i - v) \\ &= \sum_{v=v_{i-1}}^{v_i} \phi(v)(v_{i+1} - v_i)\end{aligned} \tag{1}$$

Similarly, if we insert a new v_i between two adjacent values v_{i-1} and v_{i+1}, the loss reduces by $\Delta(v_i)$, same as Equation (1).

However, because the hardware profiler only provides us the frequencies of the current *dyn_dim* value samples, we do not know the real $\phi(v)$ of every v. We make a simple assumption that the frequency distribution in each range $(v_{i-1}, v_i]$ is uniform. Then we can derive the punishment of removing each v_i based on Equation (1). We can further derive the optimal position to insert a new sample in each range, which is the center $(v_{i-1} + v_i)/2$, and the corresponding saving.

At the end of each re-sampling iteration, we need to redistribute the frequencies from the old samples to the new samples (Algorithm 1 Line 18). This procedure is shown in Algorithm 2. We iterate through all the old frequencies in `freq`. For each old frequency f for a value v_i, we identify the range $(v_{i-1}, v_i]$, i.e., $(lb, ub]$ (Lines 4 to 8), and filter the new samples falling into this range (Line 9). Following the uniform distribution assumption, we uniformly distribute f to these new samples (Lines 10 to 13).

TABLE III
HARDWARE CONFIGURATIONS OF ADYNA.

Tiles	12×12
PEs per tile	32×32
PE	FP16 MAC, 1 GHz, 64 B registers
Scratchpad	512 kB per tile, 72 MB total
Memory	6 HBM2 stacks, 1842 GB/s total
NoC	2D torus, 192 GB/s per tile

TABLE IV
AREA AND POWER BREAKDOWN OF AN ADYNA TILE.

Components	**Area** (mm^2)	**Power** (mW)
PE array	1.981	1,156.355
Scratchpad	1.413	247.927
Dispatcher + controller	0.148	10.409
Router + network interface	0.025	1.646
Total	3.567	1,416.34

VIII. METHODOLOGY

Configurations. Table III shows the default configurations of Adyna in our evaluation. The overall accelerator contains 12×12 tiles and runs at 1 GHz. All the tiles are connected by a 2D-torus-based NoC. We set each tile to have a 32×32 PE array and a 512 kB scratchpad, following [6]. We use 6 HBM2 stacks as the off-chip memory. These configurations offer 295 TFLOPs peak throughput and 1842 GB/s memory bandwidth, close to an NVIDIA A100 GPU (80 GB PCIe version). We use CACTI 7.0 [5] to model the area and power of the scratchpads. For other key components including the PE array, the kernel dispatcher, the network interface, etc., we implement RTL and synthesize them in the TSMC 28 nm technology. The RTL designs of the PE array and the router are extended from previous work [21], [68].

To analyze the dynamic operator graph and do design space exploration, we develop the model parser, the scheduler, and the cost model (Figure 4) in Rust. The generated multi-kernel schemes are then fed into a cycle-accurate simulator for hardware performance evaluation. The simulator is developed on SimPy [39] and its components are calibrated to the RTL. Re-scheduling and re-sampling are performed on the host CPU. Reconfiguration will first cause pipeline draining, whose performance impact is considered in the simulator. The reconfiguration is performed every 40 batches, so the incurred overheads are relatively low.

Workloads. The evaluated workloads are listed in Table I. The models are set up and trained using the original open-source code bases or following the instructions in the original papers. We train CV models on ImageNet [15] and NLP models on GLUE [56]. For Tutel-MoE, we implement a model similar to [41] that can be fully pipelined and fill up the on-chip buffer of a single chip. We use the test datasets in the corresponding benchmarks for evaluation and set the default batch size to 128, following [13], [66]. Other batch sizes are evaluated in Figure 13.

Baselines. The baseline M-tile accelerator in Figure 3 has the same configurations as Adyna, except for the newly proposed support for dynamic operators. The DynNN models are scheduled as static operator graphs with all the *dyn_dim* sizes set to the maximum. For M-tenant, we model Planaria [22] as a multi-tile accelerator with the same resources as Adyna. Its tiles (subarrays) can be flexibly partitioned into separate groups, while branches are modeled as individual tenants mapped to these tile groups. Switch and merge operators are done on the host CPU. We oppotimistically assume it could pre-compile [47] multiple kernels for each operator under different resource amounts, so at runtime it directly selects the best one to execute.

To illustrate the effectiveness of runtime adjustment in the scheduler, we compare with an Adyna (static) setting, which supports multi-kernel execution and dynamic data routing, uses frequency-based scheduling with an initial profiling result, but does not do re-sampling or tile sharing at runtime. To show the effectiveness of the multi-kernel re-sampling mechanism, we also use an idealized full-kernel setting, where all kernels are generated and available on-chip despite the unrealistic buffer requirement. We assume data buffering is not affected in this case, so this setting provides an upper bound for our multi-kernel scheme.

We also compare Adyna with a commodity GPU baseline of NVIDIA A100. As specified above, Adyna is configured to have similar peak FLOPs and memory bandwidth to A100 for fair comparison. To our best knowledge, most existing open-source DynNN GPU implementations either only support batch size 1 [13], [66], or simply discard redundant results without saving computations [59], [70]. Therefore, we transplant the switch (ScatterRouter) and merge (GatherRouter) implementations from Brainstorm [13], and extend the host CPU control code to support batched DynNN execution.

Adyna follows the philosophy of offline scheduling and buffering carefully selected, pre-compiled kernels on-chip. Another potential approach to supporting DynNNs is to do online real-time scheduling. We simulate this setting and compare it with Adyna. With real-time scheduling, the best matching kernel is searched every time before executing a dynamic operator. In this case, the kernel execution efficiency should be optimal, but additional online scheduling costs would occur.

IX. EVALUATION

A. Area and Power

Table IV shows the area and power of an Adyna tile. Both are primarily dominated by the PE array and the scratchpad, similar to previous DNN accelerators. To support DynNNs, the dispatcher, the controller (including the profiler), and the modified network interface logic are introduced and occupy only 4.9% chip area and 0.085% power on average. The dynamic book-keeping overheads mainly include the template kernels and the parallelization schemes of succeeding operators for dynamic routing, which are buffered in the on-chip scratchpads and occupy about 1.9% chip area and 0.088% power.

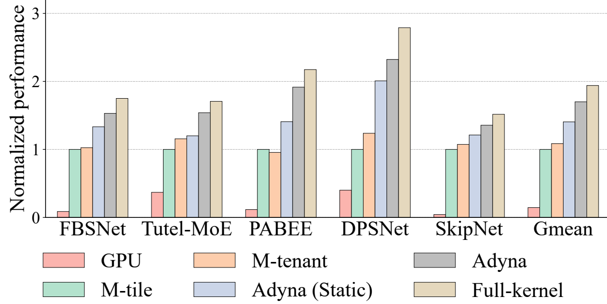

Fig. 9. Performance comparison among the GPU, the M-tile and M-tenant baselines, Adyna (static), the full-kernel setting, and Adyna.

B. Overall Performance

In Figure 9, we compare the performance among the GPU, the baseline M-tile and M-tenant, Adyna (static), the full-kernel setting, and Adyna. M-tenant is 1.09× slightly better than M-tile. It supports runtime adjustment (**F2** in Table II) to reduce unnecessary computations, but lacks on-chip inter-operator pipelining (**F3**), so it performs worse than M-tile in cases like PABEE where the intermediate tensors are large. Compared with the baseline M-tile, Adyna (static) brings an average 1.41× speedup, which indicates that supporting multiple kernels per operator and dynamic data routing in the hardware (**F4**, **F5**) can bring significant speedups. Similarly, compared with M-tenant, Adyna (static) is 1.30× faster. Then, Adyna further incorporates runtime adjustment techniques (**F2**) including real-time tile sharing and multi-kernel re-sampling to improve load balance. This brings another 1.21× speedup on average. In total, Adyna is on average 1.70× and up to 2.32× faster than M-tile, and on average 1.57× and up to 2.01× faster than M-tenant. The speedups on DPSNet are particularly high, because DPSNet folds the dynamism of other data dimensions all to the batch dimension, resulting in a very large aggregated batch size that benefits substantially from our techniques.

To further validate the effectiveness of the re-sampling algorithm, in Figure 9, we compare Adyna with the idealized full-kernel setting, which can always execute the optimal kernel for any value of *dyn_dim*. Adyna performs close to the full-kernel scheme and reaches on average 87% and up to 90% of this upper-bound performance for multi-kernel execution.

Compared with the GPU, Adyna achieves an average 11.7× speedup, which results from a joint effect including domain-specific designs, multi-tile operator pipelining, and the optimizations for dynamism in Adyna. For DynNNs that have many routing operators like SkipNet, Adyna can deliver over 30× speedups via its on-chip dynamic routing capability. Even for highly GPU-optimized Tutel-MoE, Adyna is still 4.2× faster. Considering Adyna is based on a 28 nm technology and consumes 201 W, while A100 uses 7 nm [11] and consumes 350 W, Adyna is also more energy-efficient.

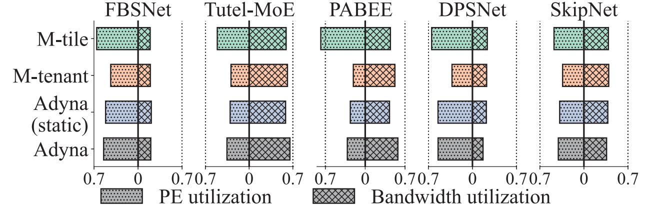

Fig. 10. PE utilization and memory bandwidth utilization of M-tile, M-tenant, Adyna (static), and Adyna.

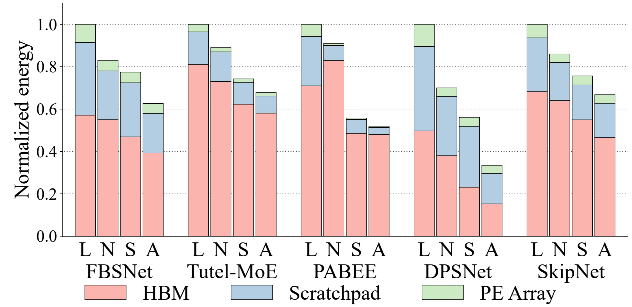

Fig. 11. Energy breakdown of M-tile, M-tenant, Adyna (static), and Adyna, corresponding to L, N, S, and A in each bar group, respectively.

C. Improvement Analysis and Breakdown

To further analyze the performance differences, we measure the PE utilization and the memory bandwidth utilization of different designs in Figure 10. First, Adyna (static) and Adyna show lower PE utilizations than M-tile. This is because they reduce the unnecessary computations in DynNNs, while M-tile always processes the worst-case maximum data sizes. M-tenant also reduces unnecessary computations, but due to the lack of inter-operator pipelining, its execution is often blocked by memory accesses, which reduces the utilization. Between the two DynNN-aware designs, Adyna has higher utilizations than Adyna (static), because Adyna eliminates a large portion of PE idleness caused by runtime workload imbalance.

Regarding the memory bandwidth, two factors contribute to the utilization differences. First, compared with M-tile, Adyna (static) and Adyna are able to reduce unnecessary off-chip data fetches when the actual *dyn_dim* is smaller than the maximum. Second, for memory-bound models like Tutel-MoE and PABEE, with profiling-guided multi-kernel sampling and dynamism-aware scheduling in Adyna, PE utilizations can increase (see above) and thus more memory bandwidth is demanded, which in turn improves the bandwidth utilization.

Figure 11 shows the energy breakdown results. Multi-kernel execution helps reduce energy consumption from all sources including memory access, SRAM access, and PE computation. In most cases M-tenant has slightly lower HBM energy than M-tile due to the elimination of redundant data accesses, while in PABEE the increased inter-operator communication outweighs the above saving. For workloads that are dominated by the HBM energy, e.g., PABEE, Adyna effectively removes

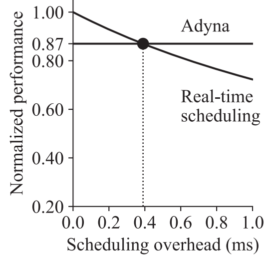

Fig. 12. Real-time scheduling overhead analysis.

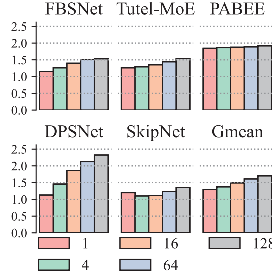

Fig. 13. Speedup of Adyna over M-tile under different batch sizes.

the redundant memory accesses; for those dominated by the on-chip PE and SRAM energy like DPSNet, Adyna removes the redundant computations and SRAM fetches. Compared with Adyna (static), runtime adjustment allows for more balanced use of scratchpads, leading to more on-chip data reuse, especially in DynNNs with more variations like FBSNet and DPSNet.

D. Discussion

Comparing to real-time scheduling. We make a comparison between the two approaches to supporting DynNNs, namely offline scheduling as in Adyna vs. online real-time scheduling. We use the performance of the full-kernel setting to simulate the optimal execution of real-time scheduling, and add extra scheduling overheads before each dynamic operator. We sweep different latencies for this online scheduling cost.

From Figure 12, as the scheduling overhead increases, the speedup of real-time scheduling vs. Adyna decreases in a nonlinear form of $T_{\text{Adyna}}/(T_{\text{opt}}+t_{\text{sched}})$ where $T_{\text{Adyna}}/T_{\text{opt}} = 0.87$ as from Figure 9. Adyna does not do scheduling on the critical path, so its performance remains unaffected. The performance cross-point of the two designs corresponds to an online scheduling latency of only 0.39 ms, i.e., only when we can make a scheduling decision within 0.39 ms, real-time scheduling could outperform Adyna. This is very difficult to achieve, if not impossible. The fastest scheduler at present, as far as we know, CoSA [27], takes about 0.1 s to produce a valid (not yet optimal) solution for an operator, implying a gap of three orders of magnitude.

Different batch sizes. Adyna mostly targets relatively large batch sizes to ensure full utilization of hardware resources. For completeness, we also evaluate smaller batch sizes. Figure 13 shows Adyna's speedups with batch sizes 1, 4, 16, 64, and 128. The average speedups are 1.29×, 1.37×, 1.49×, 1.61×, and 1.70×, respectively. As expected, since Adyna is better at alleviating larger dynamic variations caused by larger batches, the performance advantage of Adyna increases as the batch size grows, but is relatively consistent even with small batches. SkipNet is bottlenecked by the frequent off-chip transmissions of intermediate results. At batch size 1, the smaller intermediate data size allows more operators to pipeline on-chip, leading to more opportunities for Adyna's optimizations and thus higher performance.

X. Related Works

System designs for dynamic neural networks. There are several recent designs that optimize dynamic neural networks on GPUs. Brainstorm [13] alleviated dynamic load imbalance by fusing different branches and doing profiled-based speculative execution. Cocktailer [66] was a compiler that fused control flow and data flow operators in a fine-grained way to leverage low-level parallelism. However, they only supported batch-1 execution when running on a single GPU. SmartMoE [65] optimized the distributed placement of multiple experts adaptively towards the data distribution. The scheduling of Adyna also targets load balance and thus shares similar optimizations. However, since scheduling must be performed at the single-chip level instead of the system level, both the resources and the latency budget are limited.

Some of Adyna's techniques can be applied to optimize dynNNs on GPUs. For example, the dynamic operator graphs together with the profiler can be implemented in software running on the GPU, which improves load balance but has higher overheads than the hardware-assisted way in Adyna. Similarly, the multi-kernel mechanism and the kernel sampling algorithm can also be adopted by GPUs. However, as discussed in Section II-C, this software implementation is fundamentally constrained by the hardware, e.g., the limited GPU registers.

Accelerators for sparse neural networks. Among recent sparse neural network accelerators, some were for unstructured sparsity [1], [23], [48], [49], [67], e.g., SCNN [48] skipped computations when either model weights or activation values were zero. In contrast, structured sparse DNNs are usually accelerated with algorithm-hardware co-design [20], [53], [71], [72]. For example, a specially designed pruning algorithm and a tightly-coupled accelerator were proposed in DPACS [20]. For attention-based models, SpAtten [57] and Sanger [38] modeled sparse attention into sampled dense-dense matrix product and sparse matrix multiplication. Graph neural network accelerators [37], [51], [62] are proposed to deal with the sparsity and irregularity of graphs.

XI. Conclusion

We present a software-hardware co-design approach, Adyna, to accelerate dynamic architecture neural networks. Adyna uses a novel unified representation to express various DynNN types, and adopts a dynamism-aware, multi-kernel execution paradigm with hardware and software optimizations. Compared with conventional multi-tile and multi-tenant accelerators, Adyna is on average 1.70× and 1.57× faster.

XII. Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. Mingyu Gao is the corresponding author.

REFERENCES

[1] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 1–13.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016, pp. 1–12.

[3] AMD, "AI engines and their applications," https://docs.amd.com/v/u/en-US/wp506-ai-engine, 2022.

[4] AMD, "AMD XDNA™ architecture," https://www.amd.com/en/technologies/xdna.html, 2024.

[5] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, 2017.

[6] J. Cai, Y. Wei, Z. Wu, S. Peng, and K. Ma, "Inter-layer scheduling space definition and exploration for tiled accelerators," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–17.

[7] G. Chen and X. Shen, "Free launch: Optimizing GPU dynamic kernel launches through thread reuse," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 407–419.

[8] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 367–379.

[9] Z. Chen, T.-B. Xu, C. Du, C.-L. Liu, and H. He, "Dynamical channel pruning by conditional accuracy change for deep neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 2, pp. 799–813, 2020.

[10] J. Choquette, "Nvidia Hopper GPU: Scaling performance," in *2022 IEEE Hot Chips 34 Symposium*. IEEE, 2022, pp. 1–46.

[11] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany, "3.2 The A100 datacenter GPU and Ampere architecture," in *Proceedings of the 2021 IEEE International Solid-State Circuits Conference*, vol. 64. IEEE, 2021, pp. 48–50.

[12] J.-B. Cordonnier, A. Mahendran, A. Dosovitskiy, D. Weissenborn, J. Uszkoreit, and T. Unterthiner, "Differentiable patch selection for image recognition," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

[13] W. Cui, Z. Han, L. Ouyang, Y. Wang, N. Zheng, L. Ma, Y. Yang, F. Yang, J. Xue, L. Qiu, L. Zhou, Q. Chen, H. Tan, and M. Guo, "Optimizing dynamic neural networks with Brainstorm," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023, pp. 797–815.

[14] X. Dai, X. Kong, and T. Guo, "EPNet: Learning to exit with flexible multi-branch network," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 235–244.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2009.

[16] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[17] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.

[18] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "TANGRAM: Optimized coarse-grained dataflow for scalable NN accelerators," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 807–820.

[19] X. Gao, Y. Zhao, Ł. Dudziak, R. Mullins, and C.-Z. Xu, "Dynamic channel pruning: Feature boosting and suppression," *arXiv preprint arXiv:1810.05331*, 2018.

[20] Y. Gao, B. Zhang, X. Qi, and H. K.-H. So, "DPACS: Hardware accelerated dynamic neural network pruning through algorithm-architecture co-design," in *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023, pp. 237–251.

[21] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference*, 2021.

[22] S. Ghodrati, B. H. Ahn, J. K. Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *Proceedings of the 53rd International Symposium on Microarchitecture*. IEEE, 2020, pp. 681–697.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 243–254.

[24] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 11, pp. 7436–7456, 2021.

[25] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li, "FasterMoE: Modeling and optimizing training of large-scale dynamic pre-trained models," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 120–134.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[27] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "CoSA: Scheduling by constrained optimization for spatial accelerators," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*. IEEE, 2021, pp. 554–566.

[28] C. Hwang, W. Cui, Y. Xiong, Z. Yang, Z. Liu, H. Hu, Z. Wang, R. Salas, J. Jose, P. Ram, J. Chau, P. Cheng, F. Yang, M. Yang, and Y. Xiong, "Tutel: Adaptive mixture-of-experts at scale," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[29] Y. Ioannou, D. Robertson, D. Zikic, P. Kontschieder, J. Shotton, M. Brown, and A. Criminisi, "Decision forests, convolutional networks and the models in-between," *arXiv preprint arXiv:1603.01250*, 2016.

[30] L. Jarzkabek and P. Czarnul, "Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications," *The Journal of Supercomputing*, vol. 73, pp. 5378–5401, 2017.

[31] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th International Symposium on Computer Architecture*, 2017, pp. 1–12.

[32] J. D. M.-W. C. Kenton and L. K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, 2019, pp. 4171–4186.

[33] S. Kim, H. Genc, V. V. Nikiforov, K. Asanović, B. Nikolić, and Y. S. Shao, "MoCA: Memory-centric, adaptive execution for multi-tenant deep neural networks," in *Proceedings of the 29th International Symposium on High Performance Computer Architecture*. IEEE, 2023, pp. 828–841.

[34] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *Proceedings of the 52nd International Symposium on Microarchitecture*, 2019, p. 754–768.

[35] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[36] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "MISO: Exploiting multi-instance GPU capability on multi-tenant GPU clusters," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 173–189.

[37] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proceedings of the 27th International Symposium on High Performance Computer Architecture*. IEEE, 2021, pp. 775–788.

[38] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *Proceedings of the 54th International Symposium on Microarchitecture*, 2021, pp. 977–991.

[39] N. Matloff, "Introduction to discrete-event simulation and the simpy language," *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, vol. 2, no. 2009, pp. 1–33, 2008.

[40] L. Meng, H. Li, B.-C. Chen, S. Lan, Z. Wu, Y.-G. Jiang, and S.-N. Lim, "Adavit: Adaptive vision transformers for efficient image recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 12 309–12 318.

[41] Microsoft, "tutel/examples/moe_cifar10.py," 2022. [Online]. Available: https://github.com/microsoft/tutel/blob/main/tutel/examples/moe_cifar10.py

[42] R. T. Mullapudi, W. R. Mark, N. Shazeer, and K. Fatahalian, "HydraNets: Specialized dynamic architectures for efficient inference," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2018.

[43] NVIDIA, "NVIDIA multi-instance GPU user guide," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2023.

[44] NVIDIA, "Dynamic-parallelism-enabled kernel overhead," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-parallelism-enabled-kernel-overhead, 2024.

[45] NVIDIA, "NVIDIA CUDA dynamic parallelism," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism, 2024.

[46] ONNX, "ONNX operator schemas: Split," https://github.com/onnx/onnx/blob/main/docs/Operators.md#Split, 2023.

[47] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *Proceedings of the 2019 International Symposium on Performance Analysis of Systems and Software*, 2019, pp. 304–315.

[48] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th International Symposium on Computer Architecture*, 2017, pp. 27–40.

[49] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture*, 2020, pp. 58–70.

[50] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd International Symposium on Microarchitecture*, 2019, pp. 14–27.

[51] X. Song, T. Zhi, Z. Fan, Z. Zhang, X. Zeng, W. Li, X. Hu, Z. Du, Q. Guo, and Y. Chen, "Cambricon-G: A polyvalent energy-efficient accelerator for dynamic graph neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 116–128, 2021.

[52] M. Subramony, D. Kramer, and I. Paul, "AMD Ryzen™ 7040 series," *IEEE Micro*, 2024.

[53] Z. Tan, J. Song, X. Ma, S.-H. Tan, H. Chen, Y. Miao, Y. Wu, S. Ye, Y. Wang, D. Li, and K. Ma, "PCNN: Pattern-based fine-grained regular pruning towards optimizing CNN accelerators," in *Proceedings of the 57th ACM/IEEE Design Automation Conference*. IEEE, 2020, pp. 1–6.

[54] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled kernel launch for dynamic parallelism in GPUs," in *Proceedings of the 23rd International Symposium on High Performance Computer Architecture*. IEEE, 2017, pp. 649–660.

[55] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th International Symposium on Computer Architecture*, 2017, pp. 13–26.

[56] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355.

[57] H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient sparse attention architecture with cascade token and head pruning," in *Proceedings of the 27th International Symposium on High Performance Computer Architecture*. IEEE, 2021, pp. 97–110.

[58] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications," in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization*. IEEE, 2014, pp. 51–60.

[59] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, "SkipNet: Learning dynamic routing in convolutional networks," in *Proceedings of the European Conference on Computer Vision*, 2018, pp. 409–424.

[60] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[61] D. Wu, L. Pigou, P.-J. Kindermans, N. D.-H. Le, L. Shao, J. Dambre, and J.-M. Odobez, "Deep dynamic neural networks for multimodal gesture segmentation and recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 8, pp. 1583–1597, 2016.

[62] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN accelerator with hybrid architecture," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture*. IEEE, 2020, pp. 15–29.

[63] L. Yang, Y. Han, X. Chen, S. Song, J. Dai, and G. Huang, "Resolution adaptive networks for efficient inference," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2369–2378.

[64] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using Halide's scheduling language to analyze DNN accelerators," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, p. 369–383.

[65] M. Zhai, J. He, Z. Ma, Z. Zong, R. Zhang, and J. Zhai, "SmartMoE: Efficiently training sparsely-activated models through combining offline and online parallelization," in *Proceedings of the 2023 USENIX Annual Technical Conference*, 2023, pp. 961–975.

[66] C. Zhang, L. Ma, J. Xue, Y. Shi, Z. Miao, F. Yang, J. Zhai, Z. Yang, and M. Yang, "Cocktailer: Analyzing and optimizing dynamic control flow in deep learning," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023, pp. 681–699.

[67] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016, pp. 20:1–20:12.

[68] J. Zhao, A. Agrawal, B. Nikolic, and K. Asanović, "Constellation: An open-source SoC-capable NoC generator," in *2022 International Workshop on Network on Chip Architectures*, 2022, pp. 1–7.

[69] S. Zheng, X. Zhang, L. Liu, S. Wei, and S. Yin, "Atomic dataflow based graph-level workload orchestration for scalable DNN accelerators," in *Proceedings of the 28th IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2022, pp. 475–489.

[70] W. Zhou, C. Xu, T. Ge, J. McAuley, K. Xu, and F. Wei, "BERT loses patience: Fast and robust inference with early exit," *Advances in Neural Information Processing Systems*, vol. 33, pp. 18 330–18 341, 2020.

[71] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proceedings of the 51st International Symposium on Microarchitecture*, 2018, pp. 15–28.

[72] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proceedings of the 52nd International Symposium on Microarchitecture*, 2019, pp. 359–371.